Synthesis from Specifications: Basic Concepts

Frank Vahid
Sanjiv Narayan
Daniel D. Gajski

Technical Report #90-03
January 29, 1990

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

vahid@ics.uci.edu
narayan@ics.uci.edu

Abstract

The need has evolved for a synthesis tool at the computer system level. SpecSyn is one such tool. Basically, it will view the world as a set of chips communicating via protocols. Thus, an abstract specification would get synthesized into a set of one or more interconnected chips. From that point, detail is added to each chip's specification until its structure is synthesized or it is determined that a prefabricated chip similar in functionality can be used.

Features of such a tool include executable specifications from which to synthesize, constraint driven partitioning of the specifications into components (chips) and synthesis of interfaces between them, translation into VHDL and synthesis into VHDL structures of micro-architectural components, and the use of other tools (e.g. MILO, a micro-architecture and logic optimizer, and LES, a layout expert system) to evaluate the quality of the chip layout generated from VHDL description.

A major component of SpecSyn is SpecCharts, a high level specification language amenable to system level synthesis, able to represent designs from system to register transfer levels. The language consists of a hierarchy of states, represented in combined graphical and textual form, at the same time catering to the expression of concurrent behavior and specification of constraints. With it we have specified several Intel chips as well as higher level systems, and have found it to be quite powerful and easy to use.

SpecSyn will have a graphical interface, from which the user can at any time view or edit a SpecChart, translate to VHDL and simulate, view statistics provided by estimators (such as area, speed, and pins), store and retrieve SpecCharts, apply basic SpecChart operations, as well as apply the partitioning algorithms or interface synthesizer. Providing access to a wide range of tools, having a single language represent the design throughout the synthesis process, and having user specified constraints allow the user to have varying amounts of control over the synthesis process.

Contents

1 Introduction

System level synthesis refers to synthesis at the computer system level. The primary aim is to convert a system's specification into a set of one or more interconnected chips. This involves determining the number of chips necessary, distributing the specifications among the chips, finding a well-defined structure for each chip, and creating proper interchip interfaces.

Requirements of a good system level synthesis tool include:

- an executable specification language capable of representing the system at many stages of the synthesis project, with appropriate abstractions for representing a design at the system level,
- algorithms for estimations of area, speed, pins, etc.
- partitioning algorithms,
- the ability to synthesize interfaces,
- the ability to synthesize structure from a specification of a chip, and
- variable amounts of user control over the synthesis process.

This report outlines SpecSyn, a tool for system level synthesis from specifications. The report will give an overview of SpecSyn, explaining the various interactions possible between the designer and the tool, followed by a description of SpecCharts, a new executable specification language amenable to system level synthesis. A few of the major operations in SpecSyn will be discussed, including arbitration, partitioning and interface synthesis. This report also explains the concept of *protocols* as a means of simplifying the description of interprocess communication.

2 SpecSyn - A Tool for Synthesis from Executable Specifications

SpecSyn is a tool designed to aid a designer perform system level specification and synthesis. Given a specification for a multi-module system and a set of constraints, the goal is to synthesize the design into a set of interconnected chips satisfying those constraints. The SpecChart language, equipped with a wide variety of powerful features while being well suited for synthesis, provides the designer with a representation that enables him to specify a system, at many different levels, with relative ease.

An important feature of any synthesis tool should be the degree to which a designer can participate in the synthesis process. The tool should be flexible enough so as to cater to situations where in one extreme the designer may direct the entire synthesis process according to his own intuition, or in the other extreme in which the tool generates solutions automatically on request, or any situation in between. There are two general methods by which SpecSyn provides this flexibility. First, the designer can enter the initial SpecChart with varying amounts of detail. The more detailed the initial specification is, the greater is reduction in the synthesis tasks to be performed by SpecSyn. Some examples of what is meant by detail are providing the number of chips, partial or complete allocation of objects to chips, binding of states or operations to hardware components or providing low level signal transfers instead of high level algorithms.

The second method involves making all tools directly accessible to the designer. He can apply the high level automatic solution synthesizer, which uses other SpecSyn tools (like the estimator, partitioner or interface synthesizer) with the goal of satisfying the constraints of the system. He could apply the partitioner directly, choosing from different types (constructive or iterative algorithms, different algorithms among each). He could apply even lower level tools such as estimators, data structure or state relocators (manual partitioning). For every designer initiated step in the synthesis process, SpecSyn will provide the necessary follow-up actions associated with that step. For example, the designer can specify relocation of a data structure from one chip to another. In this case, the SpecSyn tool will automatically update the SpecChart, determine whether any communication is needed between chips as a result of the data structure reallocation, and insert an appropriate protocol and interface for it.

At all stages the results of the designer's actions are displayed, so he can interactively move closer towards the desired design goals. Specifically, each synthesis step is reflected in a new, possibly more detailed SpecChart. Eventually enough detail exists that one can proceed to synthesize structure for each chip.

The input to the SpecSyn synthesis system is a description of the system being designed and a set of constraints such as the number of chips, chip area or the time for executing certain tasks. The output is a set of 1 or more chips. Some of those chips may be bound to prefabricated chips, while others contain well defined hardware modules (e.g. memory, DMA controller). Figure 1 shows SpecSyn's synthesis domain. Figure 2 shows the level of possible output provided by SpecSyn. The overview of the SpecSyn tool is shown in Figure 3. We will now discuss SpecSyn's various components briefly.

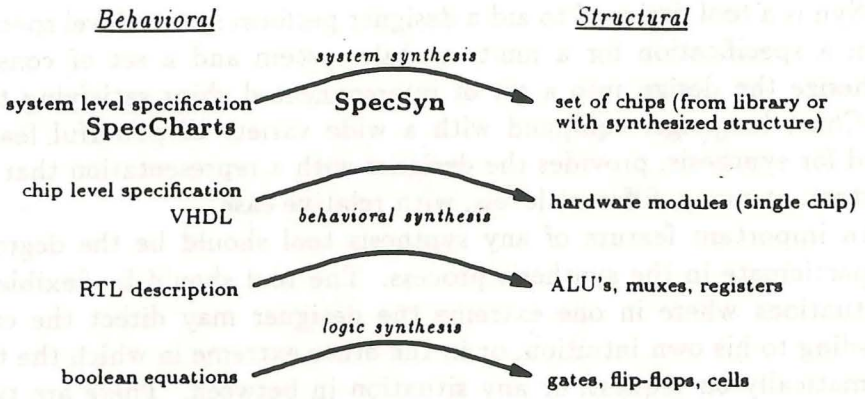


Figure 1: SpecSyn's synthesis domain

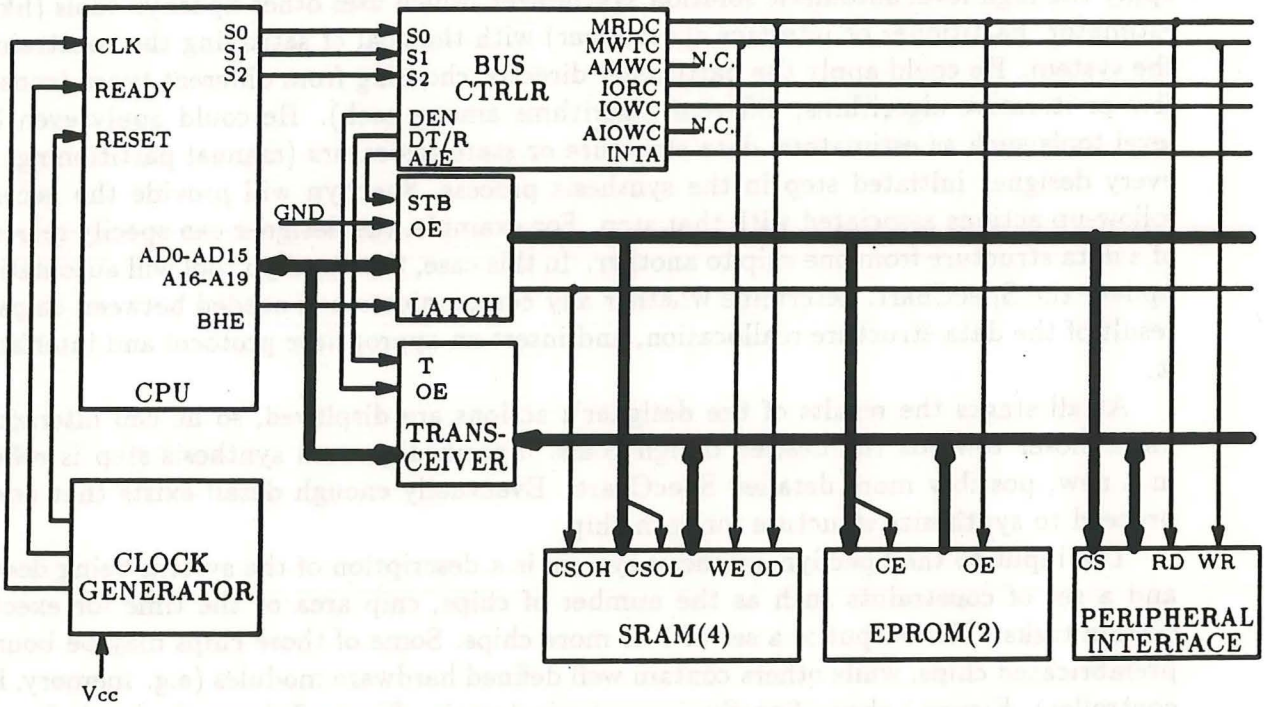


Figure 2: Sample output generated by SpecSyn

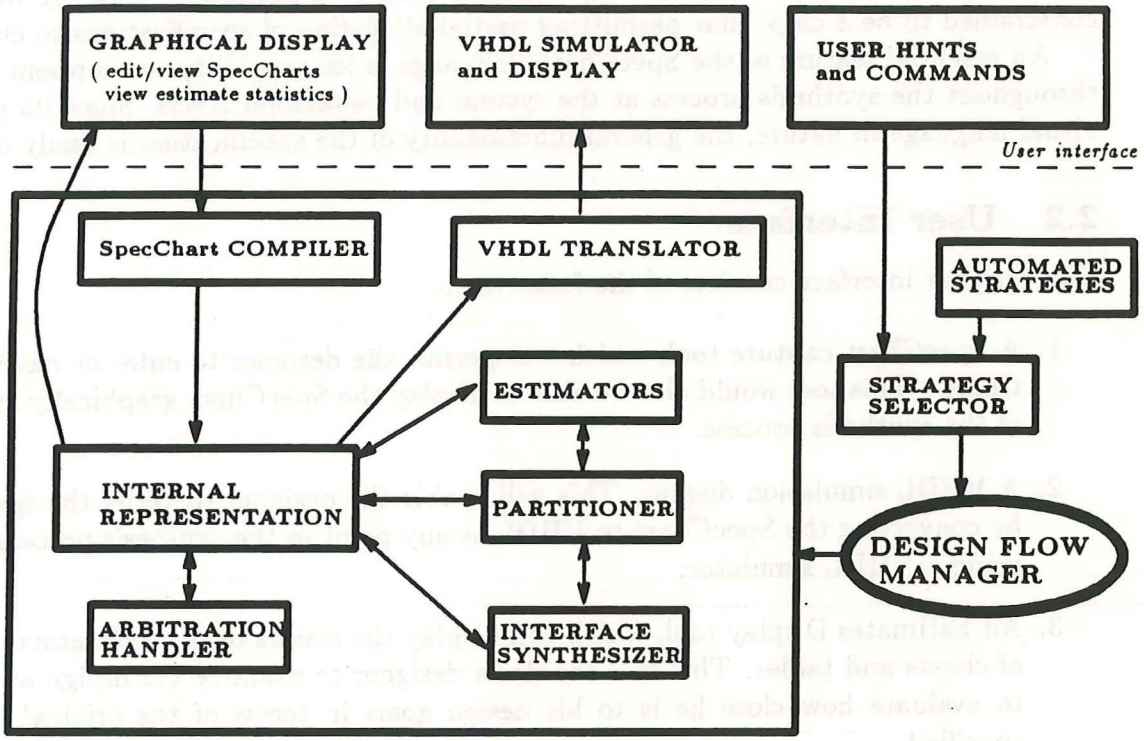


Figure 3: **SpecSyn** : An Overview

2.1 The SpecCharts Language

SpecSyn provides the designer with a representation for providing system specifications called SpecCharts. SpecCharts is a combined graphical and textual language, with the textual part being similar to VHDL. The language represents a multi-module system with a hierarchy of states, while catering to the expression of concurrent behavior and constraints. The specification may be partial or complete, and on many different levels. For example, one could specify a portion of a system using a state, providing only estimates for that state such as area or time of execution, instead of actual behavior. Or one could provide the exact behavior of that state by specifying substates, data structures, algorithms, dataflow statements, structural descriptions, or some combination thereof. A data structure could any storage element like a variable, an array or a queue. SpecCharts permits any state to be bound to hardware, such as to an adder or even to a predefined chip. A state can be constrained to be a chip, thus permitting partial allocation of specifications to chips.

An essential feature of the SpecCharts language is its capability to represent the design throughout the synthesis process at the system and behavioral levels. Since its primarily a visual language in nature, the general functionality of the specification is easily discernible.

2.2 User Interface

The designer interface consists of the following :-

1. A SpecChart capture tool, which will permit the designer to enter or modify a SpecChart . This tool would also be able to display the SpecChart graphically at any stage of the synthesis process.

2. A VHDL simulation display. This will enable the designer to verify the specifications by converting the SpecChart to VHDL at any point in the synthesis process, and then using a VHDL simulator.

3. An Estimates Display tool, which will display the results of the estimators in the form of charts and tables. This tool enables a designer to examine the design at each stage to evaluate how close he is to his design goals in terms of the original constraints specified.

4. A User Command tool, which will enable the designer to direct the synthesis process to any degree of involvement. Specific operations available to the designer include the following :-

    - obtain estimates of area, pin count, execution time
    - move data structure or processes from one chip to another
    - modify the specification chart
    - initiate optimizations like port sharing or bus creation
    - select protocols to be inserted for communication

2.3 Compilers and Translators

The SpecSyn system provides the following utilities to interface between the various tools :

- A SpecChart Compiler, which compiles the graphical/textual specification chart provided by the designer into an internal representation, and provides a preliminary check of the syntax.
- A VHDL Code Generator, which translates the internal representation to VHDL code, which can then be simulated using a VHDL simulator. At any stage of the design, a SpecChart is translatable to VHDL and then simulatable to verify the entered specification or synthesized design.
- A SpecChart Display Code Generator, which translates the internal representation back to the graphical/textual form.

2.4 Estimators

SpecSyn provides estimators for evaluating parameters like area, execution time, pin count or power consumption for any given state. The Area Estimator evaluates the area of data structures, the area needed for control logic and routing. The Time Estimator examines all paths in the state, and computes a weighted sum to obtain the average execution time. It also takes into account delays due to protocols when there are accesses to data structures in other chips. The Pin Count Estimator determines the number of pins on a chip. It not only uses port declarations, but also channel and protocol definitions to determine the number of control lines needed for interchip communication.

2.5 Arbitration Handler

Since SpecCharts permit a data structure to be accessed by many processes, contention for access to it can occur. The arbitration handler must find these situations, allow for the selection of a scheme to arbitrate between these accesses (e.g. fixed priority or rotating priority), and then select and insert proper protocols between the data structure and the accessing processes.

2.6 Partitioner

Given a specific number of chips, the partitioner must decide how to distribute objects (data structures and states) among them in order to meet the specified constraints. Several partitioning algorithms may exist. If interchip accesses are created by the partitioning, a channel for the communication must be inserted and a protocol, selected to facilitate the access, is associated with the channel. The SpecChart must then be modified to reflect the new partitioning.

2.7 Interface Synthesis

Eventually the high level concepts of channels and protocols used in a SpecChart will have to be replaced by low level items such as ports and signal assignments. This might involve matching unmatched protocols, replacing channel declarations with the port declarations specified by the definition of the protocol, replacing channel level statements like *send_data* with the actual port assignment statements, and optimizing the usage of the ports in order to reduce the number of pins.

2.8 Design Flow Manager

The Design Flow Manager is responsible for managing the SpecSyn system. It controls the flow of data across tools and the designer interface. The automated solution synthesizer decides the strategy for synthesizing from a given SpecChart and a set of constraints. The designer may also direct the synthesis process interactively. The Strategy Selector will present the Design Flow Manager with the next synthesis step to be performed, after referring to the designer directed actions and the automated solution finder.

2.9 SpecSyn Libraries

SpecSyn has several libraries to maintain several types of information. A brief description of each is given below.

- The *Working library* maintains the SpecChart of the design at successive stages of synthesis. This permits backtracking in the synthesis process, wherein we can retrace our design decisions along a backward path.

- The *Protocol library* contains predefined and user-defined protocols, which are essentially parameterized states. These protocols are used to simplify interprocess communication.

- The *Arbitration library* contains predefined and user-defined arbitration schemes which can be retrieved whenever arbitration needs to be inserted to resolve data structure access conflicts.

- The *Design Library* stores SpecCharts of all designs, which at some later point of time, may be used by the designer as part of another design. It also contains SpecCharts of prefabricated chips and components to which a designer may choose to bind a certain section of his design .

3 The SpecCharts Language

SpecCharts enable a designer to specify a multi-module system at various levels of detail. It is a combined graphical and textual representation, with the textual part being similar to VHDL. A design can be represented by SpecCharts through many stages of the synthesis process. Due to the graphical nature of the language, the general behavior of the system is easily discernible, permitting easy understanding of the design. A modification in the system as a result of a synthesis step will cause the corresponding SpecChart to be updated, and this change would be reflected in the graphical representation of the SpecChart.

In Figure 4, a block diagram of an example computer system is shown. In Figure 5, a partial SpecChart of the system is shown. At the topmost level, the specification consists of the state SYSTEM, which consists of three concurrent states - CLOCK_CHIP, KEYBOARD_INTERFACE and PROCESSOR. PROCESSOR in turn consists of a CPU similar to the Intel 8086, a DMA controller, and a memory.

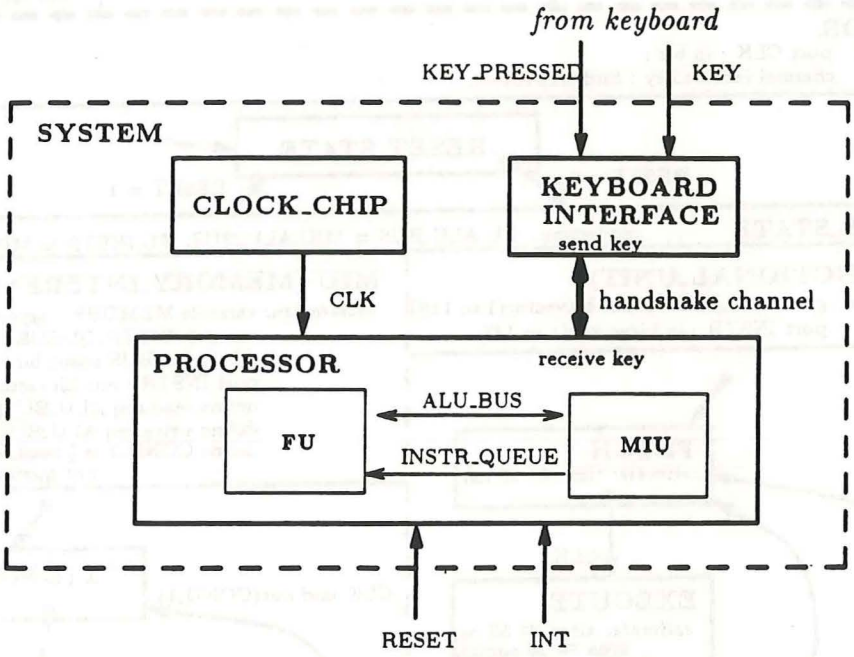


Figure 4: Block diagram of an example computer system

Since the SpecCharts description consists of a hierarchy of states, a description of the language used to represent a single state is sufficient to describe the language itself. Each state consists of the following sections:

3.1 Program Section

The Program Section specifies the actions carried out by that state (i.e. its functionality). A state can be organized in one of the three ways (Figure 6), and this is reflected in the

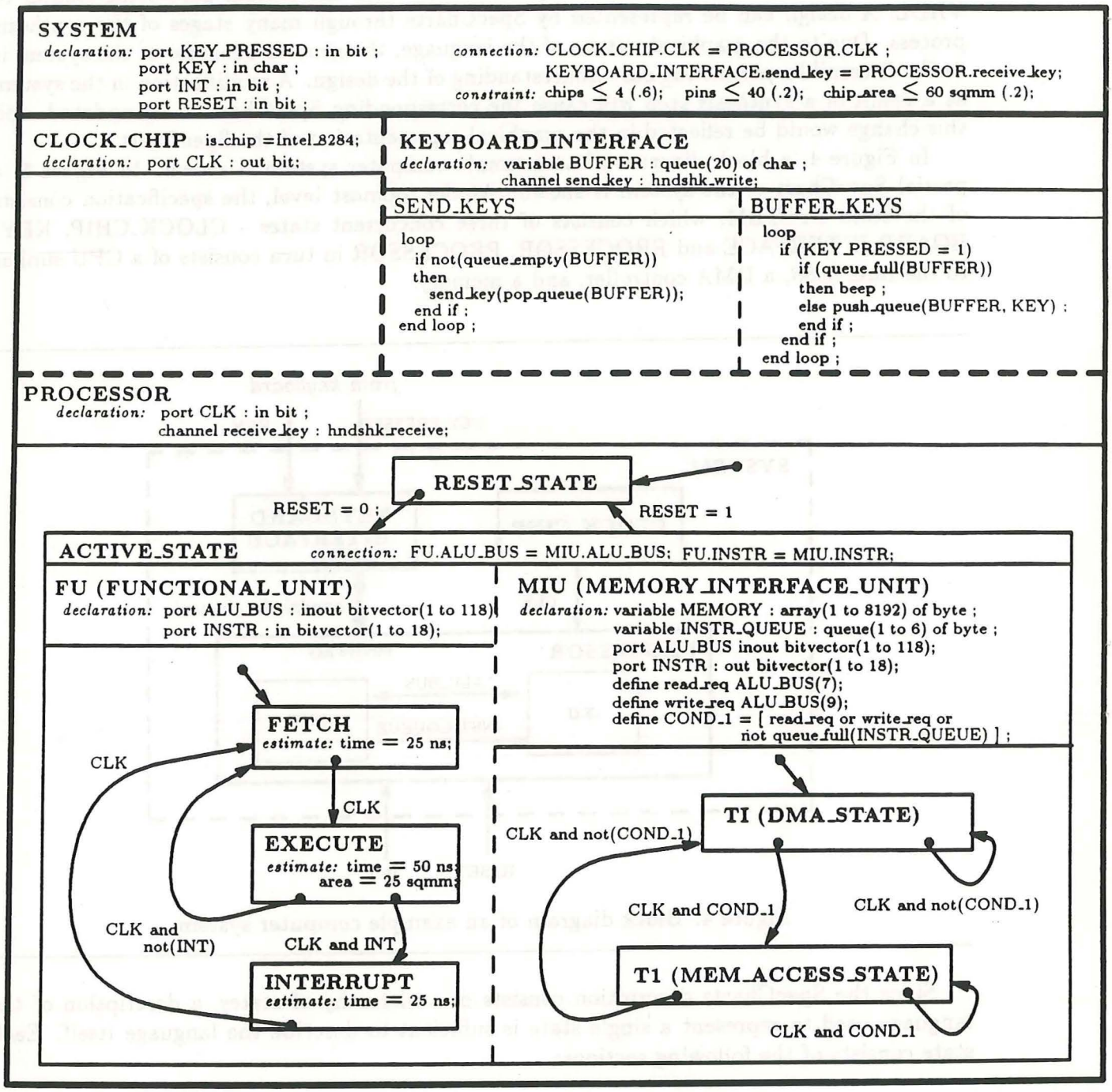
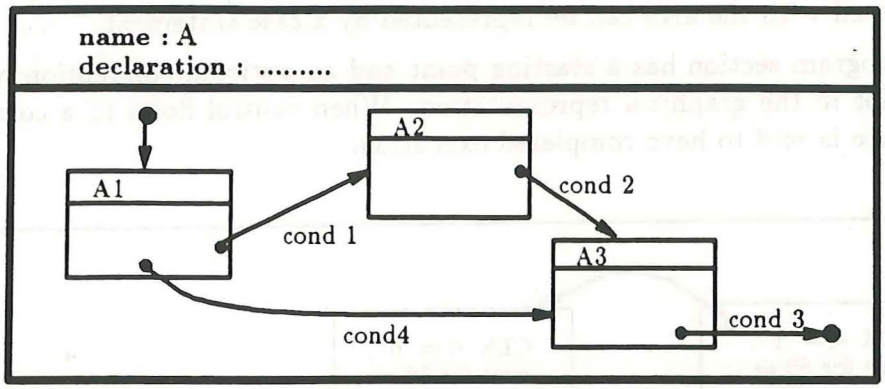


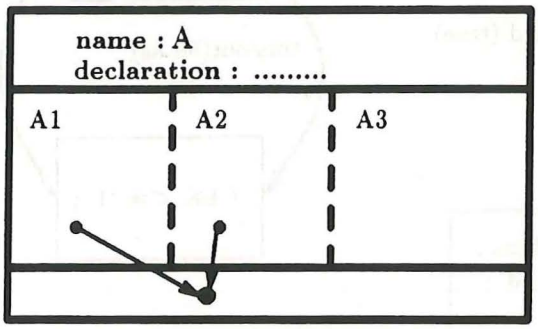
Figure 5: A Partial SpecChart of a Computer System

Program Section.

a) State A with substates sequenced by arcs



b) State A with three concurrent substates



c) State A with sequential VHDL statements

```
name : A
declaration : .....

case MODE_REGISTER is
    when '00' => OUT := 0;
    when '01' => OUT := 1;
                 RD <= '1' after 20 ns;
    when others => null;
end case;
```

Figure 6: Types of State Organizations supported by SpecCharts

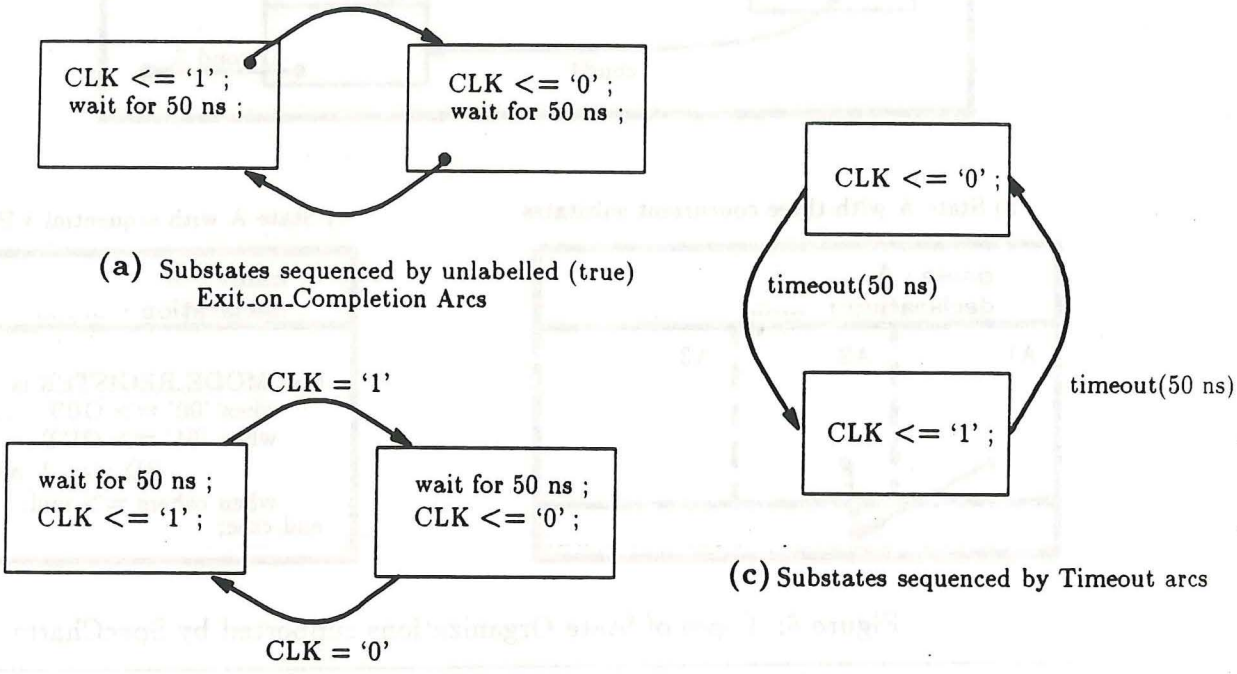
1. A state may itself consist of several states sequenced by arcs. In Figure 6a, the state A1 consists of three states (A1, A2, A3) sequenced by arcs. In Figure 5, state FU has three substates : FETCH, EXECUTE, and INTERRUPT. The substates are sequential; only one is executing at any time. In this case, the arcs between the substates indicate the order in which substates are to be executed. There are two types of arcs:

   - Exit_on_Completion arcs - indicate the next substate to be executed when the currently executing substate has completed and the arc condition is true. These arcs are shown originating from a bold dot within the substate.
   - Exit_Immediately arcs - when arc condition is true, causes the current substate's execution to terminate immediately, and execution of the state pointed to by it is

initiated. A *timeout(x)* arc is a special Exit_Immediately arc which, x time units after having entered a state, causes a transition to another state.

If at a given time more than one arc could possibly cause a transition to another state, they are assigned priority in a counter-clockwise manner, unless the designer specified a priority by numbering them. An arc with no associated condition by default assumed to be "true". A special *case arc* has been defined that can be used when the conditions associated with the arcs can be represented by a case statement.

The program section has a starting point and an optional completion point indicated by a dot in the graphical representation. When control flows to a completion point, the state is said to have completed execution.



(a) Substates sequenced by unlabelled (true) Exit_on_Completion Arcs

(c) Substates sequenced by Timeout arcs

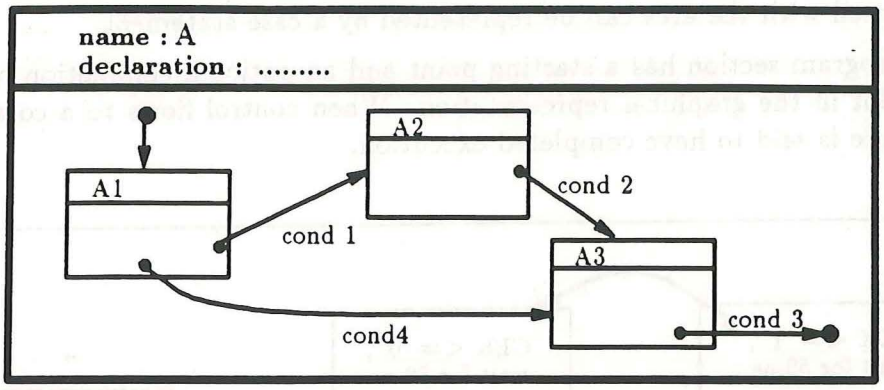
(b) Substates sequenced by Exit_Immediately arcs

Figure 7: Generation of a clock signal using different types of arcs

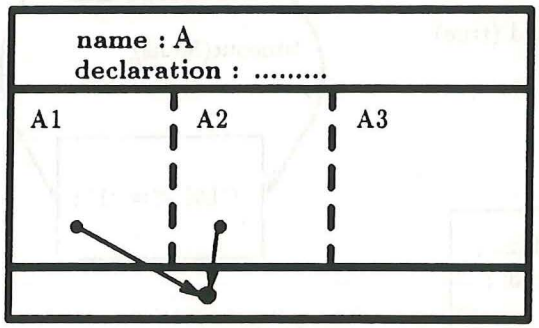
Figure 7 shows how a clock signal of period 100 ns can be generated using different types of arcs. In Figure 7a, the transitions occur after the *wait for 50 ns* statement has been executed. No condition is associated with the arc, and thus assumed to be true at all times. In Figure 7b, the last statement assigning a value to the CLK signal also triggers the exit_immediately arc transition. Figure 7c shows how timeout arcs can be used to achieve the same result. Another example of the exit_immediately arc is the arc labeled "RESET = 1" in Figure 5, which results in a transition to the RESET state from the ACTIVE state regardless of the execution status of the latter.

Program Section.

a) State A with substates sequenced by arcs



b) State A with three concurrent substates



c) State A with sequential VHDL statements

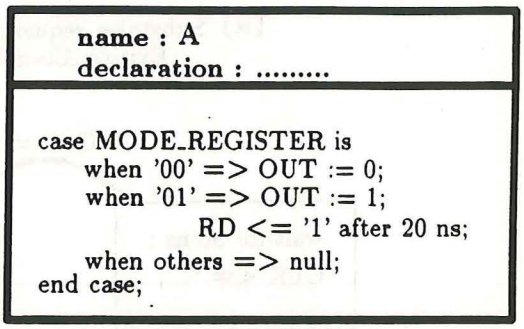


Figure 6: Types of State Organizations supported by SpecCharts

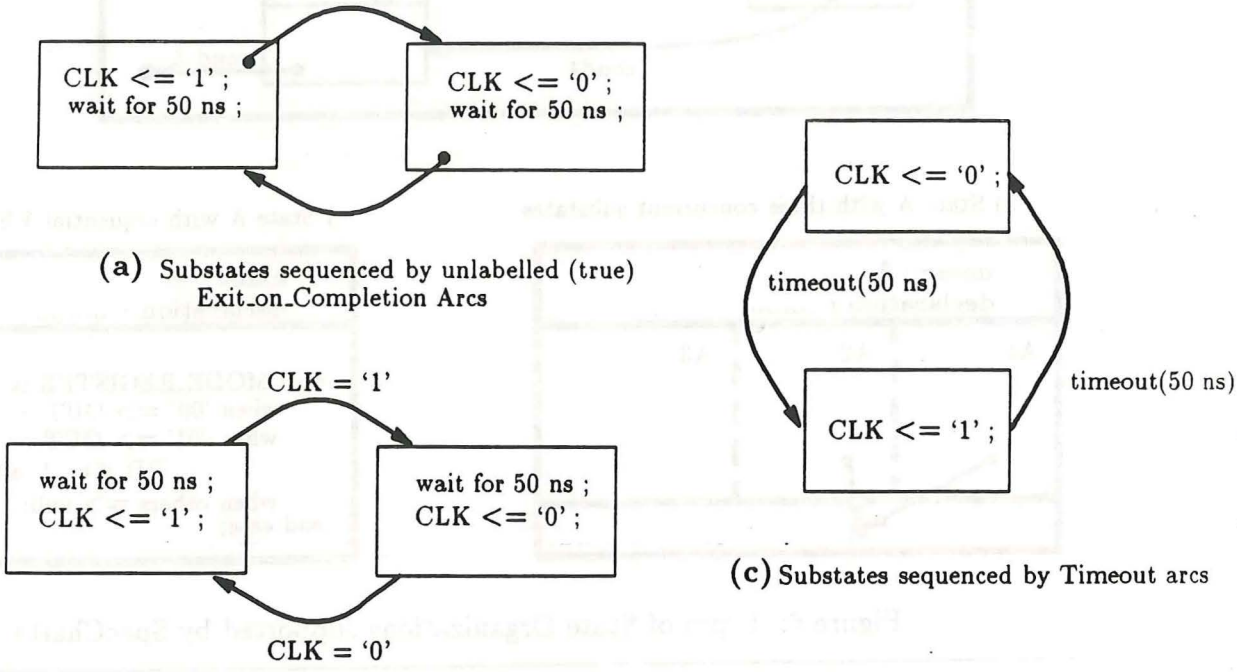
1. A state may itself consist of several states sequenced by arcs. In Figure 6a, the state A1 consists of three states (A1, A2, A3) sequenced by arcs. In Figure 5, state FU has three substates : FETCH, EXECUTE, and INTERRUPT. The substates are sequential; only one is executing at any time. In this case, the arcs between the substates indicate the order in which substates are to be executed. There are two types of arcs:

   - Exit_on_Completion arcs - indicate the next substate to be executed when the currently executing substate has completed and the arc condition is true. These arcs are shown originating from a bold dot within the substate.
   - Exit_Immediately arcs - when arc condition is true, causes the current substate's execution to terminate immediately, and execution of the state pointed to by it is

initiated. A *timeout(x)* arc is a special Exit_Immediately arc which, x time units after having entered a state, causes a transition to another state.

If at a given time more than one arc could possibly cause a transition to another state, they are assigned priority in a counter-clockwise manner, unless the designer specified a priority by numbering them. An arc with no associated condition by default assumed to be "true". A special *case arc* has been defined that can be used when the conditions associated with the arcs can be represented by a case statement.

The program section has a starting point and an optional completion point indicated by a dot in the graphical representation. When control flows to a completion point, the state is said to have completed execution.



(a) Substates sequenced by unlabelled (true) Exit_on_Completion Arcs

(c) Substates sequenced by Timeout arcs

(b) Substates sequenced by Exit_Immediately arcs

Figure 7: Generation of a clock signal using different types of arcs

Figure 7 shows how a clock signal of period 100 ns can be generated using different types of arcs. In Figure 7a, the transitions occur after the *wait for 50 ns* statement has been executed. No condition is associated with the arc, and thus assumed to be true at all times. In Figure 7b, the last statement assigning a value to the CLK signal also triggers the exit_immediately arc transition. Figure 7c shows how timeout arcs can be used to achieve the same result. Another example of the exit_immediately arc is the arc labeled "RESET = 1" in Figure 5, which results in a transition to the RESET state from the ACTIVE state regardless of the execution status of the latter.

2. A state may consist of concurrent substates (often referred to as processes). Each process is a state by itself, except that all of them are initiated when the parent state starts executing. The parent state is said to terminate when all or some (as indicated by the designer) of the processes have finished execution. Graphically, concurrent processes are shown separated with dotted lines. Thus in Figure 6b, the state A consists of three concurrent substates (A1,A2,A3), and the designer has specified that A terminates when both A1 and A2 have terminated. In Figure 5, the state SYSTEM consists of three processes - CLOCK_CHIP, KEYBOARD_INTERFACE, and PROCESSOR..

   There are four methods by which processes can communicate with sibling processes (see Figure 8):

   (a) using global data structures (declared by an ancestor state),

   (b) with direct access to a sibling's data structures by specifying *state_name.ds_name*,

   (c) through ports,

   (d) through channels.

   During synthesis, the first two methods will result in creation of channels. Likewise, channels will be converted to ports during interface synthesis. Thus, all the methods will eventually be converted to ports at the appropriate stage of the synthesis process, in order to achieve a detailed specification.

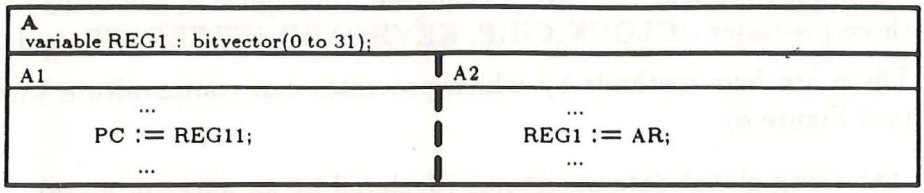
   For both this and the previous program organization, if the parent state is terminated (either by its parent terminating or by an exit_immediately arc), all its executing substates and processes are also terminated.

3. The state may be a *leaf* state, meaning it has no substates or subprocesses. Such a state has sequential VHDL code only. The VHDL statements may not include an entity, architecture, process, configuration, component, procedure or function declarations. It may only contain code that could appear within a VHDL process. This organization is show in Figure 6c. The KEYBOARD_INTERFACE state in Figure 5 has two concurrent substates, each of which are leaf states containing VHDL statements.

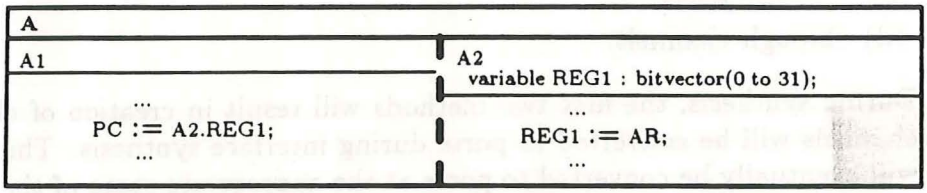
3.2 Name Section

This section contains the name of the state, certain optional attributes, and any parameters. An example of an attribute is *is_chip*. It indicates that the state comprises a single chip. The designer can allocate all objects to 'is_chip' states, and then check constraints or repartition. Or, the designer could partially allocate the objects to chips. Either way, with the 'is_chip' attribute the designer can provide a varying amount of guidance to the partitioner. Eventually during synthesis all objects will be descendants of 'is_chip' states, with the goal that the partitioning meets constraints. Note that for obvious reasons, no descendant of an 'is_chip' state may be an 'is_chip' state.

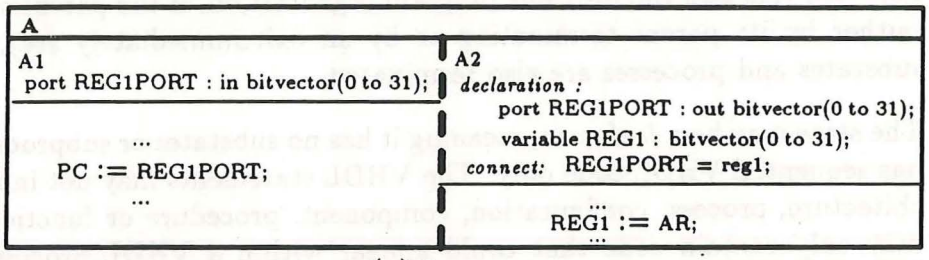
Another attribute is shown as = *component_name*, which binds the state to the indicated prefabricated component whose SpecChart and layout are found in a library.



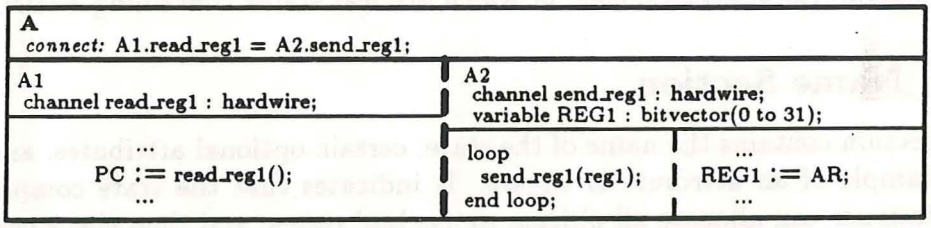
(a) Shared data structure



(b) Direct sibling data structure access



(c) Ports



(d) Channels

Figure 8: Four methods of interprocess communication

A third attribute is 'is_ports_only_module', indicating that the state cannot access any items declared in parent states through scoping rules, i.e. it is a module that can only be communicated with via ports. The previous attribute implies this attribute.

In Figure 5, the designer has indicated that CLOCK_CHIP is bound to the prefabricated Intel 8284 Clock Generator chip, and consequently no synthesis is performed on this portion of the system. No other states have been allocated to any chips at this point. During synthesis, more 'is_chip' states will need to be added; certainly more than one, since a CPU, DMA controller, memory, and keyboard interface probably will not fit on a single chip.

States can have parameters, similar to those allowed for macros. The parameterized state can then be stored in Design library for use by other SpecCharts as a substate or even a procedure call. Each reference in another SpecChart is expanded using the previously stored state definition, and parameter replacement is done by name, similar to that in C macros.

3.3 Declaration Section

This section specifies the data structures, macros, procedures, functions, types, ports, and channels whose scope includes all descendant states (all substates at any depth of hierarchy).

Data Declarations in SpecCharts consist of the following :-

- Literals, types, subtypes as in VHDL
- Data structures supported by SpecCharts are variables, arrays, records, stacks and queues. In addition to the standard VHDL operation set, SpecCharts allow operations associated with stacks and queues such as POP_STACK or PUSH_QUEUE. Certain bit manipulation operations like SHIFT and ROTATE are also provided.
- Attributes associated with classes of items are also featured as in VHDL. Moreover additional attributes such as STACK_FULL or QUEUE_EMPTY are defined in the language.
- Procedure and Function declarations, and aliases similar to VHDL
- User defined macros, such as those use in C. eg. #define READ read_req and read_enable

Ports are one way to achieve interprocess communication (see concurrent substates in program section below for other ways). Any two processes with the same parent state can communicate through ports. Thus in Figure 5, the MEMORY_INTERFACE_UNIT and the FUNCTIONAL_UNIT of the PROCESSOR communicate with each other over an ALU_BUS. Thus, ALU_BUS is declared as ports in both the processes. Port declarations consist of the following parameters:

- Name of the port
- Direction of data through the port, i.e. in, out, or inout
- Type of port, i.e. integer, real, bit, tristate etc.
- (optional) Active level, i.e. high, low, high/low

- (optional) Class i.e. data, control or clock.

Channels are also declared in the declaration section. Channels are a high level abstraction used to avoid having to specify low level ports and data transfer statements for interprocess communication. The channels are declared in the declaration section and have a protocol associated with them. This permits the designer to specify communication using high level descriptive channel names like *send_data* or *read_memory*. The protocol associated with the channel determines the exact details of the communication. In Figure 5, a channel is declared for the data transfer between the KEYBOARD_INTERFACE and the PROCESSOR. A *handshake* protocol is associated with the channel. A channel declaration consists of the following:

- channel name
- protocol to be used

The SpecCharts language recognizes a few standard protocols such as hardwire, handshake, and fixed time demand protocols. The designer may also define his own protocols. Protocols are considered in detail in section 4.

3.4 Connection Section

The Connection Section specifies port and channel connections between the following :

- Two communicating subprocesses,
- The ports/channels of a state with ports/channels of its children states, or
- The ports/channels of a state with other ports/channels of the same state.

Figure 5 has a connection specified for the ALU_BUS ports on the FU and the MIU, whereas a channel connection is specified in the state SYSTEM, to connect PROCESSOR.RECEIVE_KEY with KEYBOARD_INTERFACE.SEND_KEY.

3.5 Constraints Section

The Constraint Section is used to optionally specify design constraints such as area, pin_count or the execution time of a state. The designer may specify the relative importance of these constraints; this will influence decisions made during synthesis. The constraints not only guide the partitioner but also enable knowing when a satisfactory design has been achieved. For example, the constraints specified in the state SYSTEM in Figure 5 specify that the design has to be synthesized into at most 4 chips and each chip can have no more than 40 pins or an area greater than 60 sqmm. The relative importance of these constraints is also indicated as 0.6, 0.2 and 0.2 respectively.

3.6 Estimates Section

It may be the case that in the initial stages of the design process, the designer decides not to specify the internal details of some (or all) states, but still wants to perform some of the higher level synthesis tasks (e.g. partitioning). To provide for this very likely scenario, SpecCharts has an Estimates Section. This section enables the designer to specify his own estimates of certain quantities of the state, such as area, time, and data structure usage, instead of the actual contents of the state. Usually these quantities would be determined by the estimator by using the details specified in the Declaration and Program Sections. If the Estimate Section exists, the estimator will simply use the values specified therein. This enables the designer to start with a very coarse specification and add more and more detail at various stages of the design process. For example, in Figure 5, the designer has omitted the details of the FETCH state, but has indicated that it takes 25 ns to execute.

4 Protocols - A Mechanism for Interprocess Communication

4.1 General Description

A process may access data structures which may be associated with another process. The designer will have to provide a description of the access by specifying the data transfers and the necessary synchronization. Interprocess communication can occur very frequently in a design and a single access may consist of several operations which have to be specified by the designer every time an access is made. To enable the designer to concentrate more on the design than having to specify the details of interprocess transfers repeatedly, SpecCharts provide the concept of *protocols*, which are a defined method of data transfer.

A *channel* is a high level abstraction of interprocess communication, wherein every access is treated as a transfer of data over the channel between the two processes. A channel is a combination of ports at the process boundaries and a protocol. To facilitate such accesses, a channel must be declared and the appropriate protocol associated with it. In the program section, for each interprocess access, the designer simply has to specify the channel and the data structure(s) involved in the transfer.

Specifying a protocol for interprocess accesses identifies not only the mechanism of communication but determines the ports that must be declared for each of the interacting processes. Ports have to be declared both for the data transfers and the necessary control lines. In case the two processes which communicate with each other exist on two different chips, the ports associated with the protocol are the actual *pins* of the respective chips. During interface synthesis, the low-level data transfer and control statements will be generated according to the protocol definition.

4.2 Classification of Protocols

A library of those protocols which are frequently used to facilitate accesses to a single data structure is already built into the SpecChart language. These protocols may classified according to several criteria :

- Data structure accessed : If the data structure involved is an array then an address might also be supplied in order to access an element of the array. If it is not an array, no address need be supplied. The class of protocols which are used to access arrays are called ***address*** protocols. An example of an address protocol is accessing data from a memory, where the address of the word being accessed must be specified.

- Time involved in access : In *hardwired* protocols, the value of the data structure is available at all times at the ports of the *requesting* chip, and consequently no delay is involved in reading the data structure. The clock signal from a clock generator to the various chips in a system is an example of a hardwired protocol. A *fixed-time* protocol has a predetermined time interval between the requesting of an access and the granting of the access. An example of this protocol type is a CPU reading the data from the memory by placing the address on the address bus and activating the 'RD' line, and

receiving the data from the data bus after a time interval equal to the access time of the memory. Another class of protocols is the *handshake* protocol, where the access to a data structure is granted by an explicit acknowledge signal in response to a request for access. In this case, the time needed to access a data structure varies from access to access. Interrupt driven I/O involves a handshake between the I/O device and the processor using the interrupt and interrupt acknowledge signals.

- Direction of transfer : A protocol may be further classified depending on whether it is employed to read from or write to a data structure.

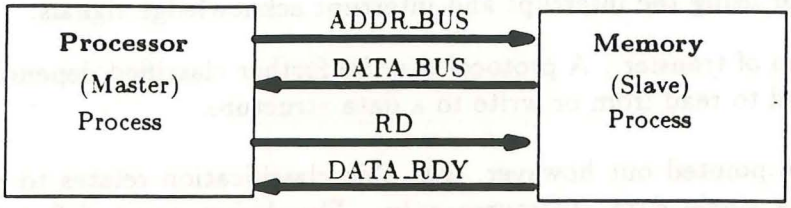
It must be pointed out however, that this classification relates to protocols which are used to access single data structures only. The designer can define arbitrarily complex protocols using SpecCharts, which may involve accesses to several data structure and even have computations performed on them.

4.3 Specification of Protocols

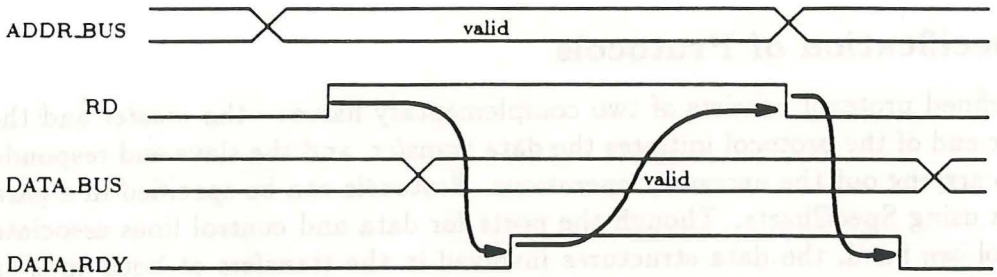
Each predefined protocol consists of two complementary halves - the *master* and the *slave*. The master end of the protocol initiates the data transfer, and the slave end responds to the master by carrying out the necessary operations. Protocols can be specified in a parameterizable form using SpecCharts. Though the ports for data and control lines associated with the protocol are fixed, the data structures involved in the transfers at both ends can vary with the instantiation of the protocol at each interprocess access.

Figure 9 defines an *Address Handshake* protocol. In Figure 9a, the block diagram of the protocol with the master end being a processor, and the slave end being a memory, is shown. The data and control lines required by the protocol are also shown - ADDR_BUS, DATA_BUS, RD, and DATA_RDY. In Figure 9b, a traditional timing diagram representation is shown for the protocol. In Figure 9c, the two complementary halves of the protocol - Addressed Handshake Read and Addressed Handshake Send are shown as parameterizable SpecChart states, with the actual low-level data transfer statements that constitute the protocol. The italicized parameters will be specified at each access point. Thus the master process has to specify an *addr_register* and *data_register*, whereas the slave process will specify the *array* involved in the protocol. A channel has to be declared on both the processor and the memory, and the protocols associated with each.

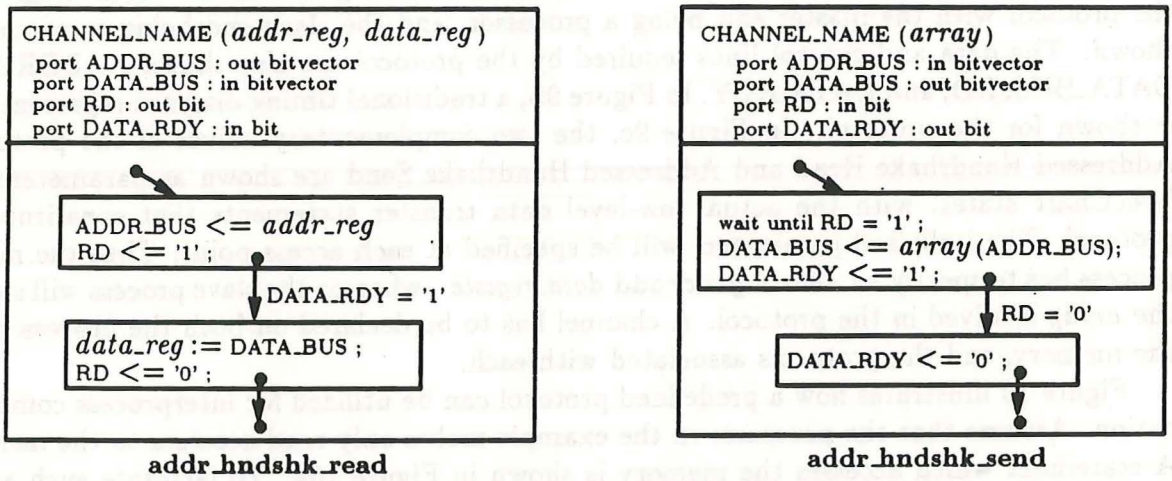
Figure 10 illustrates how a predefined protocol can be utilized for interprocess communication. Assume that the processor in the example makes only read accesses to the memory. A statement which accesses the memory is shown in Figure 10a. To facilitate such an access, a channel is declared in both the processor and the memory and the *address handshake* protocol associated with it. The channel name can be any identifier, and in this example we have selected descriptive names such as READ_MEMORY and SEND_DATA. The statement " INSTR_REG <= MEM(PC) " gets replaced by the parameterized instantiation of the protocol. This is shown in Figure 10b. Finally, interface synthesis will use the definition of the protocol to expand the channel references by the actual low-level data transfer statements. The ports associated with the protocol are also added in the declaration section of the state. The expansion of the master process (i.e. processor) is shown in Figure 10c.



(a) Block Diagram for addressed handshake read

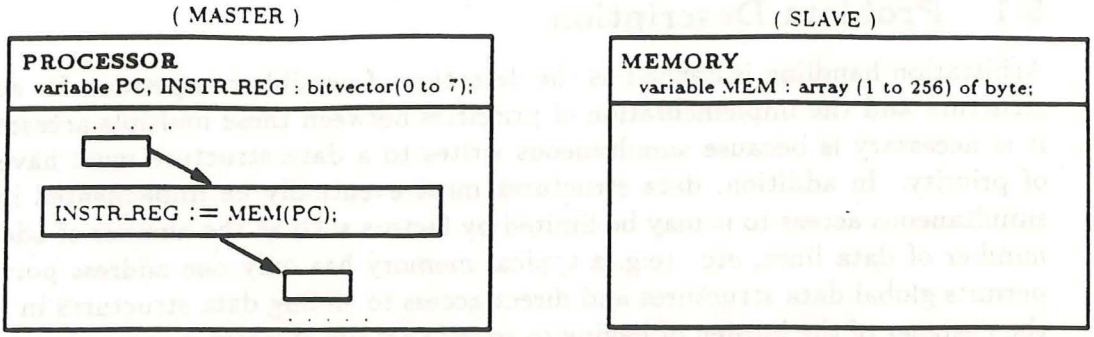


(b) Traditional timing diagram representation of the addressed handshake read protocol

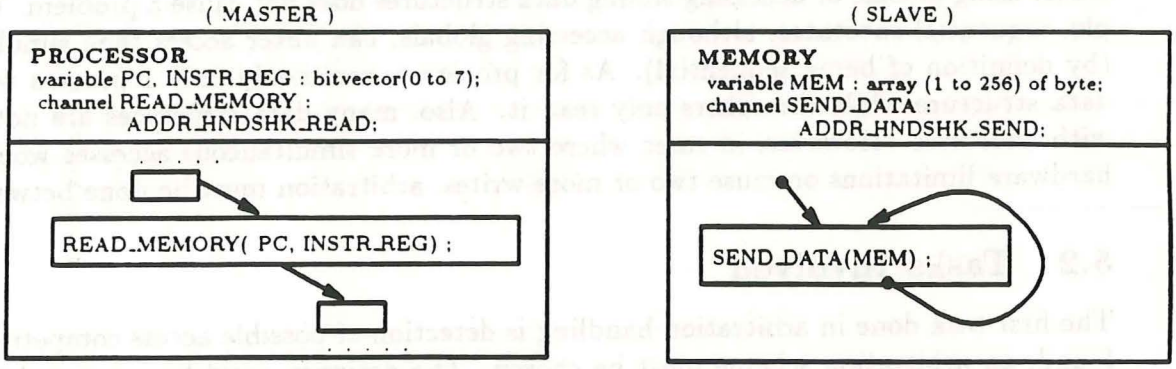


(c) SpecChart representation of the addressed handshake read and send protocols (protocol parameters in italics)

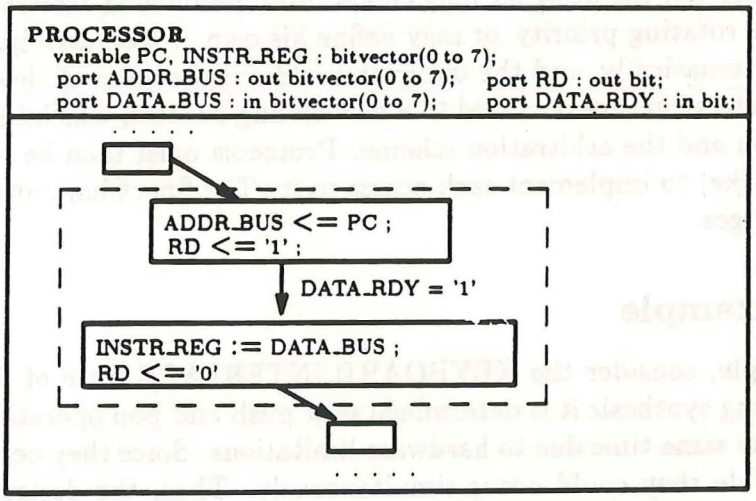
Figure 9: Representation of the Address Handshake Protocol



(a) Diagram showing inter-process access of data-structure MEM



(b)Diagram showing channel declaration and parameterized protocol instantiation



(c) Diagram showing protocol expansion with actual parameters and ports after interface synthesis

Figure 10: Protocol instantiation and expansion

5 Arbitration Handling

5.1 Problem Description

Arbitration handling is defined as the detection of possible competition for access to a data structure and the implementation of priorities between these multiple accesses. One reason it is necessary is because simultaneous writes to a data structure must have a some order of priority. In addition, data structures must eventually be implemented in hardware, so simultaneous access to it may be limited by factors such as the number of address ports, the number of data lines, etc. (e.g. a typical memory has only one address port). SpecCharts permits global data structures and direct access to sibling data structures in order to relieve the designer of the burden of having to specify details of interprocess communication. These features create the possibility of simultaneous access to the same data structure. In most cases, using globals or accessing sibling data structures does not cause a problem. For example, sequential substates, although accessing globals, can never access then simultaneously (by definition of being sequential). As for processes, commonly only 1 process writes to a data structure while the others only read it. Also, many data structures are not accessed with addresses. However, in cases where two or more simultaneous accesses would exceed hardware limitations or cause two or more writes, arbitration must be done between them.

5.2 Tasks Involved

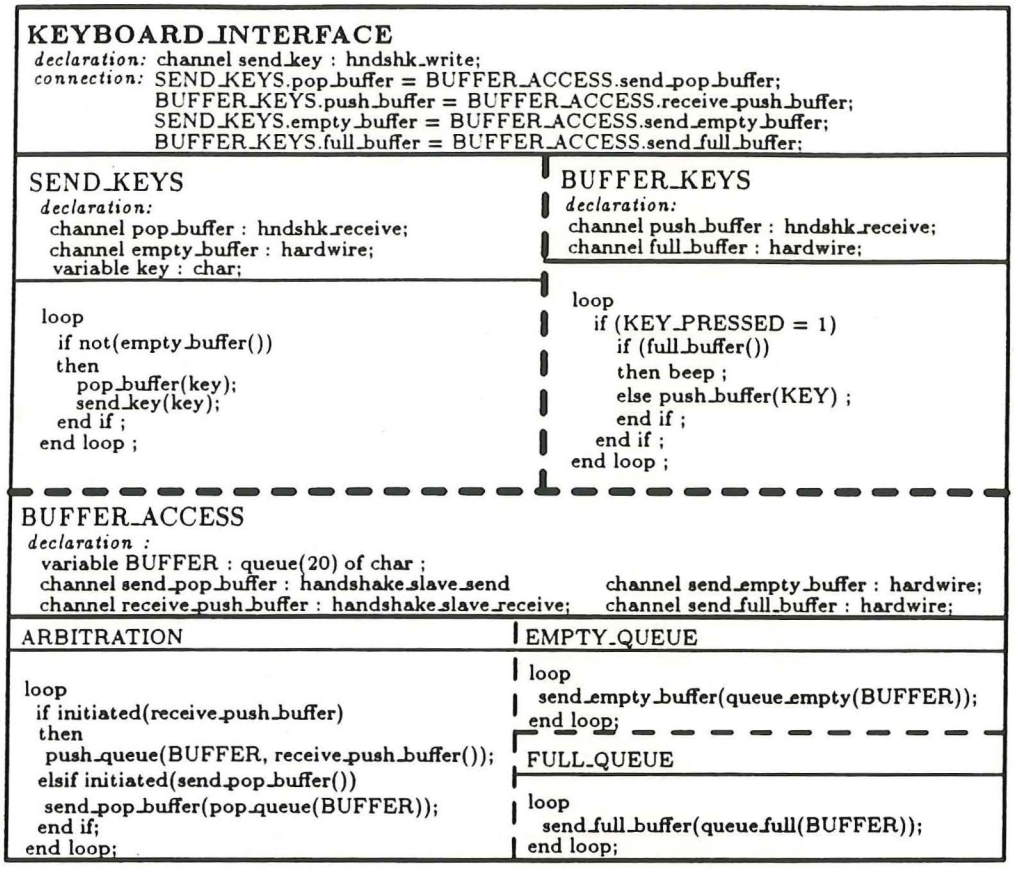
The first task done in arbitration handling is detection of possible access competition. Once found, an arbitration scheme must be chosen. The designer could be prompted to provide a scheme. He may state that one is not necessary if he knows for sure that a conflict will never occur. Or, the designer may choose from predefined arbitration schemes, such as fixed priority or rotating priority, or may define his own. Alternatively, a default scheme could be chosen automatically, and the designer could change some of these later.

The data structure is moved to a new sibling process, which includes the data structure's declaration and the arbitration scheme. Protocols must then be selected (usually some form of handshake) to implement each access to it. The SpecChart must be updated to reflect all these changes.

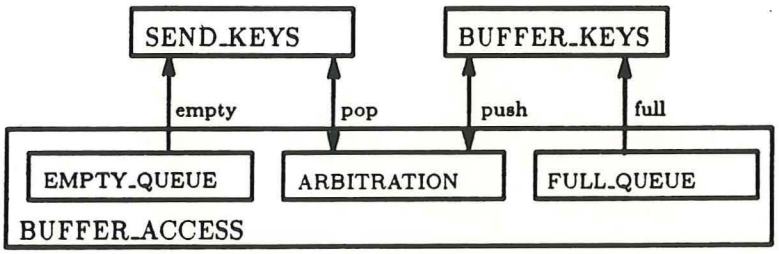
5.3 Example

For example, consider the KEYBOARD_INTERFACE state of Figure 5. Suppose at some point during synthesis it is determined that push and pop operations on BUFFER cannot be done at the same time due to hardware limitations. Since they occur in concurrent substates, it is possible they could occur simultaneously. Thus, the designer is prompted to provide an arbitration scheme. He may create his own by entering a SpecChart implementing the arbitration, or he may choose from several predefined schemes. Suppose he chooses a fixed priority scheme with the push operation having higher priority. BUFFER is then moved to a new process, BUFFER_ACCESS, which includes the arbitration SpecChart.

All accesses to BUFFER will now be made through protocols. Since the push and pop operations may have to wait an unknown amount of time, handshake protocols are chosen.



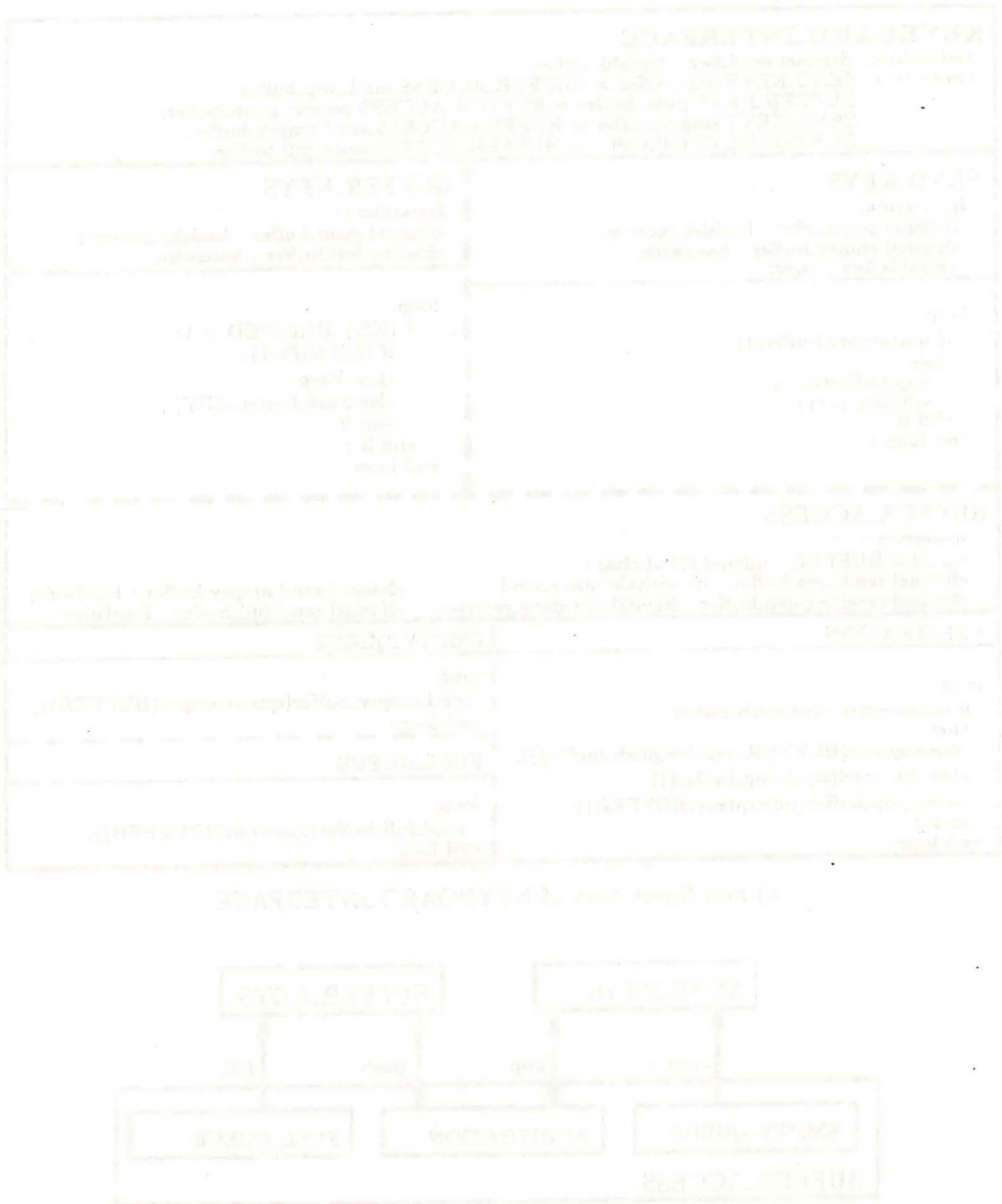
a) new SpecChart of KEYBOARD_INTERFACE



b) block diagram of KEYBOARD_INTERFACE

Figure 11: Updated KEYBOARD_INTERFACE state after arbitration

The queue_empty and queue_full tests can be replaced by a simple hardwire protocol , since they do not compete with push or pop, and moving them to BUFFER_ACCESS will not add a delay when they are accessed by the pop and push processes. The modifications to the KEYBOARD_INTERFACE state are shown in Figure 11.



6 Partitioning

Partitioning is defined as the distribution of objects among chips, where an object is a data structure or a state. Partitioning is done to try to meet the specified constraints, e.g. area per chip, pins per chip or the execution time of processes. There are two major tasks associated with partitioning: *choosing the partition*, and *implementing the partition.*

6.1 Choosing the Partition

There are two major types of partitioning algorithms, *constructive* and *iterative.* Constructive partitioning starts with a partial partitioning, in which some or all objects are not allocated to chips, and finds a complete partitioning, in which all objects are allocated to chips. Iterative partitioning attempts to improve upon a complete partition (tries to better meet the constraints).

Constructive Partitioning – The goal is to create an initial partitioning of the larger objects of the design, so that iterative partitioning can concentrate on fine tuning for constraints, without having to make major changes.

Iterative Partitioning – The goal is to repartition an existing partition to come closer to or meet the desired constraints. This task can be broken up into three phases:

1. Selecting objects to move - perhaps each object is a candidate, or each pair of objects, or perhaps even random selection (e.g. simulated annealing).
2. Determining where to move the candidate objects - some methods are group migration, pairwise exchange, simply moving candidates to chips giving best score for a single move or even random movement (such as simulated annealing).
3. Scoring - to determine how much of an improvement occurs in meeting constraints for a particular set of moves, a 'score' is needed to rate each partitioning. It may be based on estimations of area, pins, speed, or some combination thereof.

A synthesis tool should not restrict the designer to one type of algorithm; rather, he should have access to several different algorithms. In addition, combinations of algorithms should exist, e.g. first apply a constructive, then an iterative algorithm. The designer can choose an algorithm or algorithms to meet his requirements, which may depend on the size of the design, the stage in the design process, the desired quality of the design, and the time available to develop the design.

6.2 Implementing the Partition

Once a partition has been chosen, the SpecChart must be updated to reflect it while still maintaining the same basic functionality. The three main tasks include *finding interchip accesses*, *selecting protocols* and *updating the SpecChart.*

Finding Interchip Accesses — The new partition may create interchip accesses, the access by a state on one chip to a data structure on a different chip. The first step is thus to find all such accesses.

Selecting Protocols — For each such access, a protocol must be selected to facilitate the data transfer.

Updating the SpecChart — Finally, the SpecChart can be updated to reflect the changes. First we move each object to its new chip. For each interchip access we declare channels with the protocols determined above, and then connect the channels. Each data structure access statement is then replaced by a channel call such as 'read_memory'. The chip containing the data structure being accessed gets processes added that contain corresponding data receive or send statements.

6.3 Example

Consider the system in Figure 5. Intuitively, we note that a CPU, DMA controller, and an 8K memory probably won't fit on a single chip. In Figure 12, a certain stage of synthesis is shown. At this stage, there are 3 chips, and all objects except KEYBOARD_INTERFACE have been allocated to a chip. The diagram shows an example manual partitioning session.

The automatic partitioner could have arrived at the same partitioning. For example, a particular partitioning algorithm might first allocate all unallocated objects (constructive), so the KEYBOARD_INTERFACE gets placed on CHIP2. Then, suppose CHIP1's area is estimated at 135 sqmm (square millimeters). This violates the 60 sqmm chip_area constraint. A particular iterative algorithm may decide that MEMORY should be moved to CHIP2. This is very likely because the partitioner is constraint driven, so that this redistribution would greatly improve our area-per-chip situation without worsening pins-per-chip or time too much. On the other hand, the extremely large increase in pins (ports) that would occur if we moved FU to CHIP2 gives it a very poor chance of being moved, even though with respect to area alone it is a very good move.

Now that a new partitioning has been chosen (either manually or automatically), it must be implemented. The first step is to find all interchip accesses. In Figure 13a an access to MEMORY by T1 is shown. Since the access involves providing an address and then waiting for data, the protocol chosen is a type of addressed handshake.

The SpecChart can now be updated. MEMORY is moved to CHIP2, and channels are declared. The access to MEMORY by CHIP1 is replaced with a *READ_MEMORY* statement that initiates the data transfer, and a process is added to CHIP2 that sends the data when requested (slave). The relevant changed portions are shown in Figure 13b.

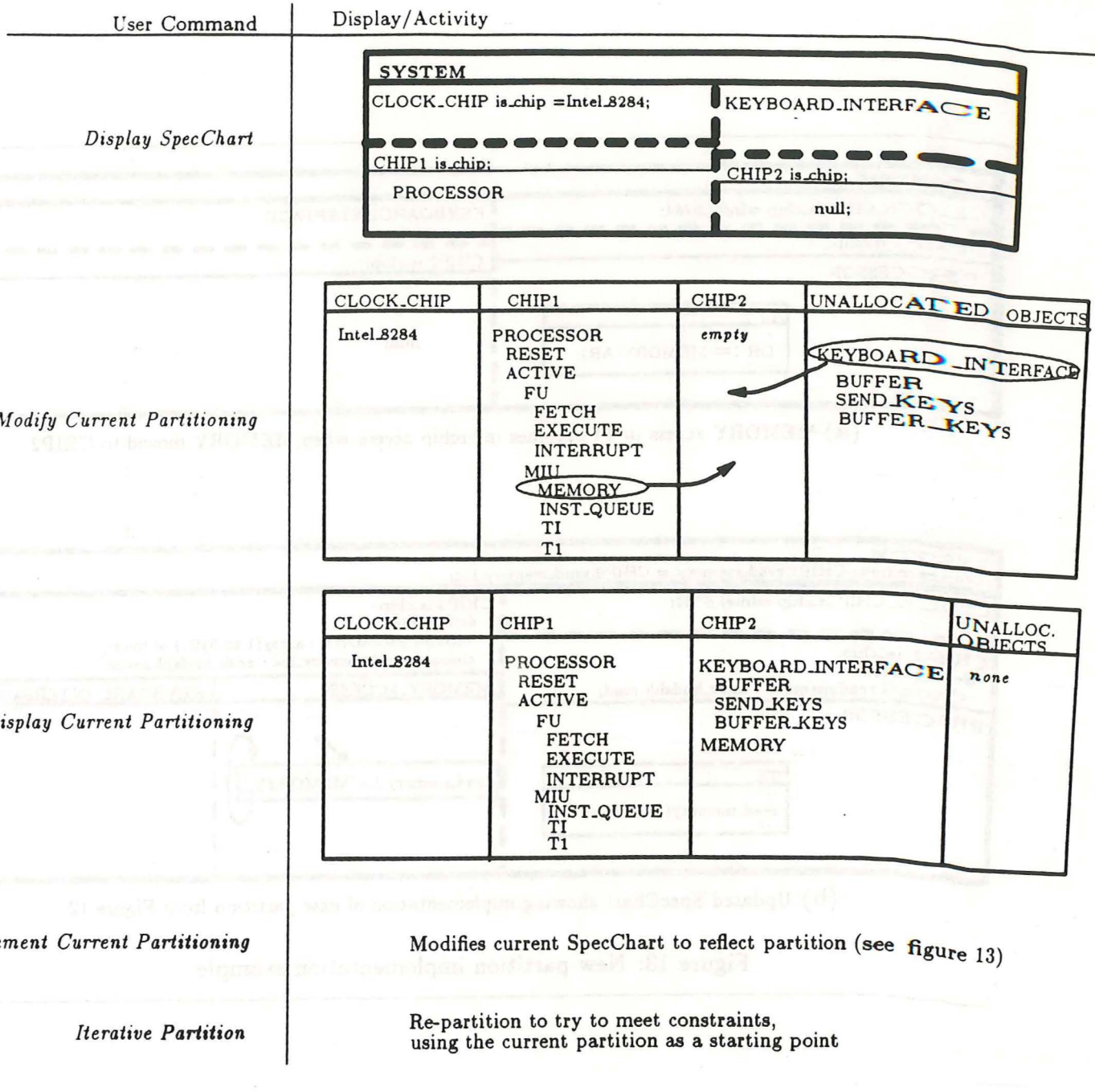
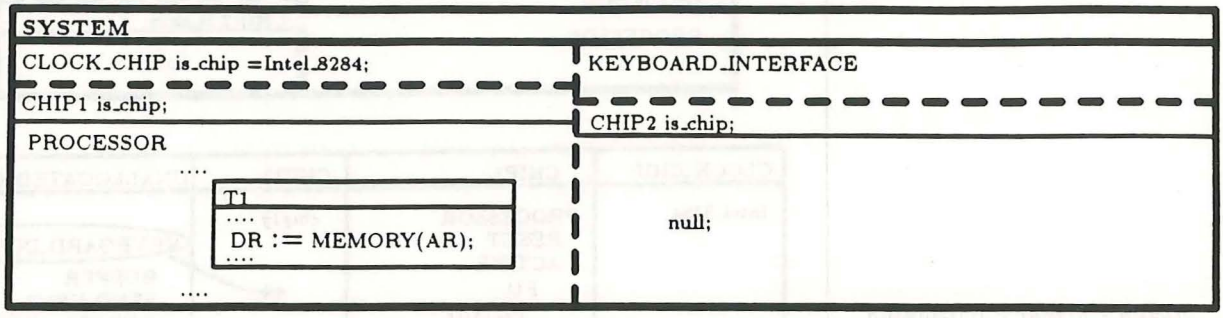
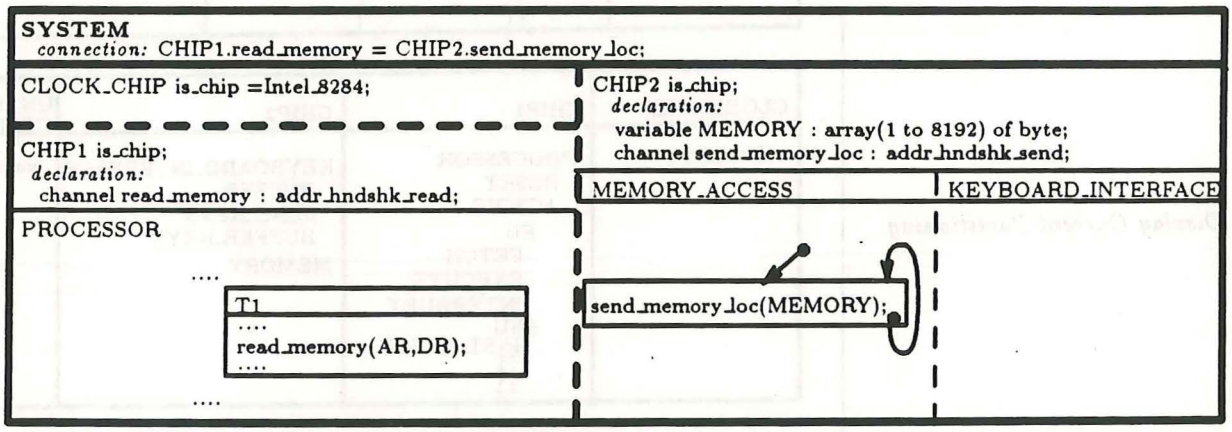


Figure 12: Partitioning example showing possible manual partitioning



(a) MEMORY access in TI becomes interchip access when MEMORY moved to CHIP2



(b) Updated SpecChart showing implementation of new partition from Figure 12

Figure 13: New partition implementation example

7 Interface Synthesis

Interface synthesis is defined as synthesizing channels and protocols into ports and low-level signal assignments that implement a data transfer. Channels and protocols are abstract concepts used to hide the details of interprocess data transfer. This greatly simplifies the specification, but at some point the lower level details may need to be synthesized, e.g. when synthesizing structure for the chip, passing a detailed specification to some other tool, simulating, or estimating time or pin count more accurately. There are 3 major tasks associated with interface synthesis: *protocol matching, synthesis of port and signal assignments*, and *pin optimization.*

7.1 Protocol Matching

It is possible for two channels to be connected that do not have matching protocols. For example, one channel might be a handshake while the other a fixed-time protocol. Unmatched protocols can occur when the protocols were specified by the user, or when trying to use a prefabricated chip to implement part of a SpecChart. The interface synthesizer should either modify one of the protocol declarations or add glue logic so that the data transfer is made successfully. For example, if one protocol specifies the sending of two 8-bit pieces of data and the other a receiving of one 16-bit piece of data, an 8-bit latch might be inserted and control signals modified.

7.2 Synthesis of Port and Signal Assignments

The task of synthesizing ports and signal assignments from channels and protocols is greatly simplified since protocols themselves are represented as SpecCharts. If this was not the case, the actual signal assignment statements would have to somehow be figured out from the protocol representation. In our case, the protocol is 'inserted' into the SpecChart where the send or receive was, with a few small naming details taken care of. Thus, for the simple protocols predefined so far, the synthesis is essentially done by macro expansion.

7.3 Port Optimization

It might be possible to reduce the number of ports (and thus pins) by performing some analysis of the port usage. For example, ports might be merged if determined that their usage is exclusive in time, or if the pin constraint is exceeded but the speed constraint is not.

7.4 Example

In the section on Partitioning, the example given involved the insertion of channels and protocols in order to implement a data transfer, specifically a read from MEMORY. After interface synthesis, channels and protocols are replaced with ports and low-level signal assignments. Figure 14 shows the relevant part of the example's SpecChart after interface

synthesis. Note the difference between this figure and Figure 13b. There is substantial savings in hiding the low level details with channels and protocols, especially when one considers a design might have 10 or 15 channels, not just 1 as in this example. Not only does it allow the designer to concentrate on the higher level aspects of the design, it also permits easier change. For example, to change a protocol from a handshake to a fixed-time one, the user simply changes the declaration.

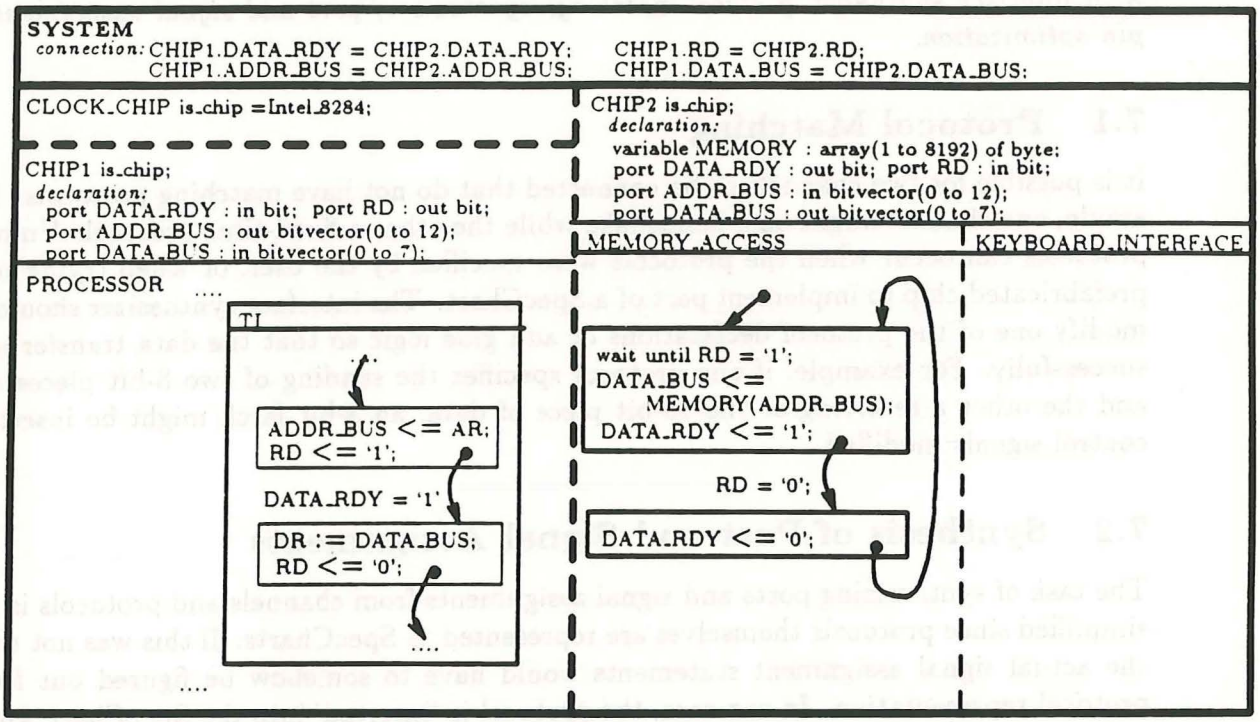


Figure 14: MEMORY access after interface synthesis

Although not discussed in the partitioning example, it is certain that the MIU does not only read from MEMORY, but also writes to it. The write would have been given a separate channel, and had an addressed handshake master send protocol associated with it. However, this uses more pins than is necessary, since the MIU never tries to read and write at the same time. Thus, the address ports for each channel can be shared, as can the data lines. The port optimization phase would detect this, merge the ports, and update the SpecChart appropriately.

8 Summary

Specification at the system level requires that less detail need exist in the description. It is the job of synthesis to provide these details. SpecSyn provides the SpecCharts specification language to represent the design at both system and register transfer levels. It is a combined graphical/textual language, permitting partial specification, expression of constraints, concurrency, hierarchy, and structure. It is also simulatable.

Tasks involved in system level synthesis include partitioning, arbitration handling, interface synthesis, and estimation. SpecCharts use protocols and channels to represent data transfer at an abstract level, thus greatly simplifying specifications.

Currently, we have defined the major tasks of system synthesis and have refined each task into further subtasks, while developing preliminary algorithms for several of these. We have defined the SpecCharts language, and modeled several Intel chips using it, and have found the language to be powerful method of specifying systems at any level of detail.

Future work includes implementing some examples and refining the algorithms. The SpecChart representation will be implemented and the VHDL translator built. We can then begin to implement the actual synthesis.

9 Acknowledgements

This work was supported by the Semiconductor Research Corporation under contract #89-DJ-146. We are grateful for their support. We would also like to thank Joe Lis and Tedd Hadley for their helpful suggestions.

References

[Br88] Brewer, F.D., "Constraint Driven Behavioral Synthesis", Ph.D. thesis, University of Illinois at Urbana Champaign, May 1988.;

[Ha87] Harel, D., "Statecharts : A Visual Formalism for Complex Systems" , Science of Computer Programming 8, 1987 pp 231 – 274. ;

[Intel89] Intel Corporation, "Microprocessor and Peripheral Handbook, Vol. 1 & 2 ", Intel Corporation, 1989

[LiGa88] Lis, J., and Gajski, D., "Synthesis from VHDL", ICCD, 1988.;

[LiSU89] Lipsett, R., Schaefer, C.F., and Ussery, C. "VHDL : Hardware Description and Design " Kluwer Academic Publishers, 1989. ;

[PrLo88] Preas, B. and Lorenzetti, M. "Physical Design Automation of VLSI Systems ", Benjamin/Cummings Publishing Company Inc., 1988. ;